Autocomplete Dictionary System

Vikrant Satheesh Kumar (vs2778) and Aswin Tekur (at3584)

Contents

1. Introduction

The Autocomplete dictionary system aims to complete broken words based on the k-th most frequent word from its dictionary if it exists. If there is a frequency tie, then any of the matching words can be returned to complete the broken word.

For example, if the frequency dictionary is built based on the following (k=1):

- Harry Potter books:
  “You’re a wiz Har” → “You’re a wizard Harry”

- The Matrix movie script:
  “He’s beginn to bel” → “He’s beginning to believe”

- Dude Where’s My Car movie script:
  “Dud where’s m c” → “Dude where’s my car”

2. Implementation

2.1 Sequential Autocomplete

A. The word frequency dictionary is built using the given input file filtering the words.

B. This dictionary is used to build a trie data structure which stores the word along with its frequency to lookup and complete words.

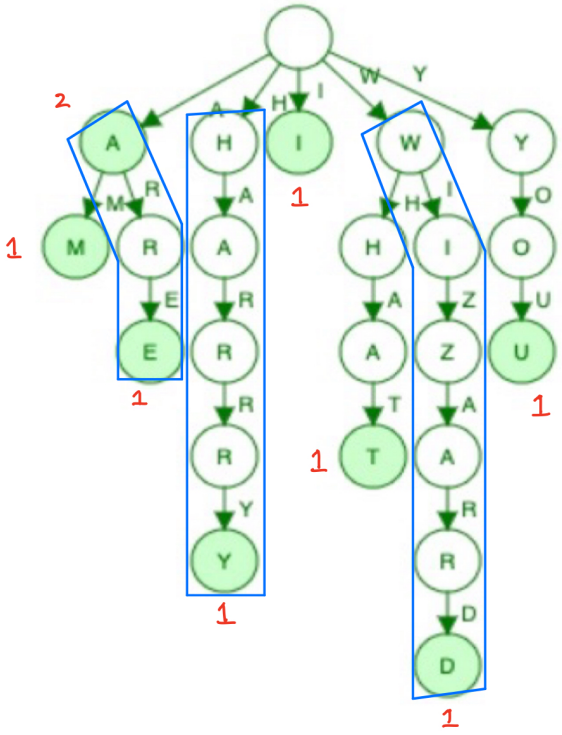
C. The broken words to be filled are read from a file and completed using the k-th most frequent word from the trie that matches the word pattern.

Example:
Dictionary built with: *"You are a wizard Harry. I am a what?"*
Broken sentence: *"You ar a wiz har"*
Completed sentence: *"You are a wizard harry"*

A. Word Frequency Dictionary

you : 1	harry : 1
are : 1	I : 1
a : 2	am : 1
wizard : 1	what : 1

B & C. Trie with word frequency



A. Building the word frequency dictionary

1. Words are filtered based on alphabetical presence and converted to lowercase.

```
wordCleaner :: [[Char]] -> [[Char]]
wordCleaner x = L.map (L.map toLower . L.filter isAlpha) x
```

Time taken to print cleaned words for different file sizes.

File Size	Time
5.8 MB	0.682 s
84.1 MB	8.122 s
161.2 MB	19.580 s
322.4 MB	41.957 s

2. Word frequency map built with cleaned up words.

Approach 1: Inserting into a map and folding over it

```
mapWordsSeq :: (Foldable t, Ord k) => t k -> Map k Int
mapWordsSeq = L.foldr (\v s->insertWith (+) v 1 s) M.empty
```

Time taken to print frequency map of words for different file sizes on cleaned words.

File Size	Time
5.8 MB	2.135 s
84.1 MB	26.689 s
161.2 MB	67.220 s
322.4 MB	136.660 s

Approach 2: Mapping each word to 1 and mapping to add the counts at the end

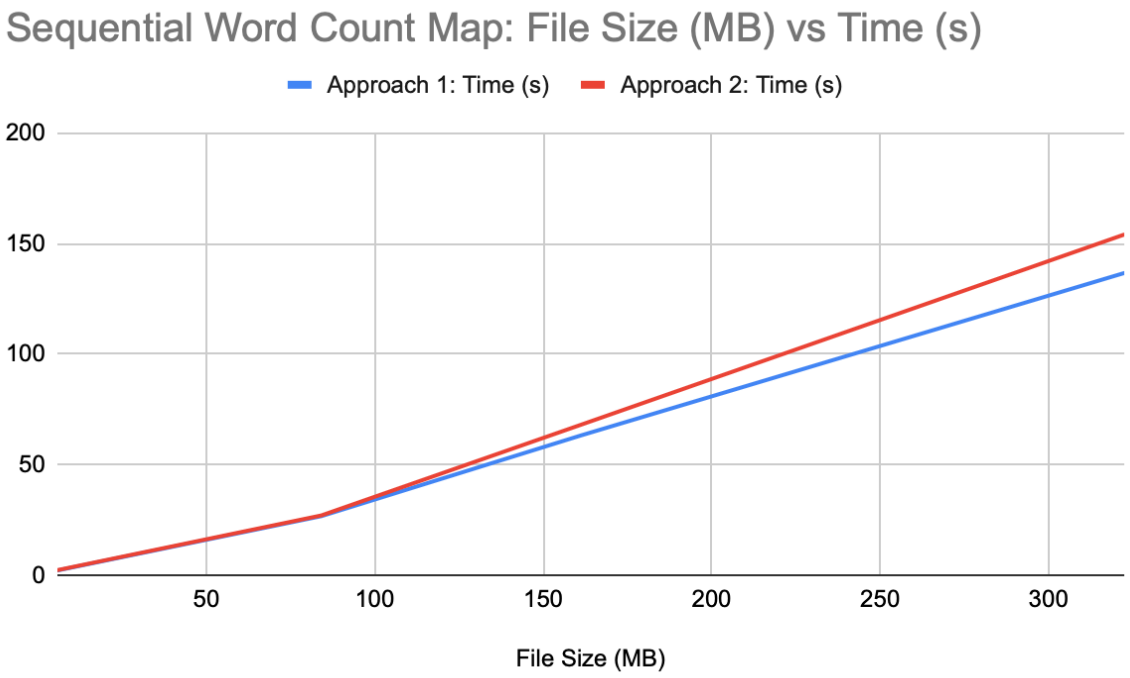
```
mapWords :: [String] -> [(String, Int)]
mapWords cw = L.map (\x -> (x,1)) cw
```

```
mapReduce :: (Ord k) => [(k, Int)] -> Map k Int
mapReduce x = (M.fromListWith (+) x)
```

Time taken to print frequency map of words for different file sizes for the cleaned words.

File Size	Time
5.8 MB	2.270 s
84.1 MB	27.017 s
161.2 MB	68.055 s
322.4 MB	154.060 s

Comparing Approach 1 vs Approach 2



Approach 1 is faster than approach 2 based on the data to build the word frequency map sequentially.

B. Building the word trie

Takes a list of words with its respective frequency and constructs a trie using Data.Trie.

```
trieBuilder :: [(String, Int)] -> Trie Int
trieBuilder w = T.fromList (L.map (\x -> (C.pack(fst x), -(snd x))) w)
```

Time taken to print trie for different file sizes with frequency map built using approach 2.

File Size	Time
5.8 MB	2.442 s
84.1 MB	28.716 s
161.2 MB	70.680 s
322.4 MB	140.560 s

C. Word Complete

Returns the k-th most frequent word that matches a given word pattern from the trie if present, else returns the word itself.

```
getK :: Ord b => [(C.ByteString, b)] -> Int -> [Char] -> [Char]
getK t k w = kWord
   where
       kWord = if length l > k then C.unpack (fst (l !! k !! 0)) else w
       l = groupBy (\x y -> snd x == snd y) (L.sortBy (O.comparing snd) t)
```

Given a word, checks to see if the word is already complete, in which case it returns the word itself. Otherwise, returns the k-th most frequent word that matches the word pattern in the trie if present, else returns the given word without any change.

```
wordComplete :: Ord a => [Char] -> Int -> Trie a -> [Char]
wordComplete w k trie = kWord
   where
       resTrie = T.toList (T.submap (C.pack w) trie)
       notPresent = isNothing (T.lookup (C.pack w) trie)
       kWord = if (length resTrie > 0 && notPresent) then getK resTrie k w
else w
```

Given a list of words, complete it with the k-most frequent word if possible.

```
completeWords :: Ord a => [[Char]] -> Int -> Trie a -> [[Char]]
completeWords ws k trie = [wordComplete w k trie | w <- ws]
```

Dictionary File Size	Broken File Size	Time
5.8 MB	0.004 MB	2.235 s
21.2 MB	0.016 MB	8.009 s
42.2 MB	0.18 MB	21.906 s
74 MB	0.32 MB	42.687 s

2.2 Parallel Autocomplete

A. After the words have been filtered in parallel chunks, the list of cleaned words is split up into multiple chunks using which multiple frequency maps are built in parallel. These maps are further merged together in parallel until the number of maps is reduced to a smaller extent after which it is merged sequentially to build the final word frequency map.

B. After the word count is obtained, the map of the word count is converted into a list which is broken down into chunks. These chunks are used to build individual word frequency tries in parallel. These individual tries are merged together in parallel summing up common word frequencies until the number of tries are reduced to a smaller extent after which it is merged sequentially to build a singular word frequency trie.

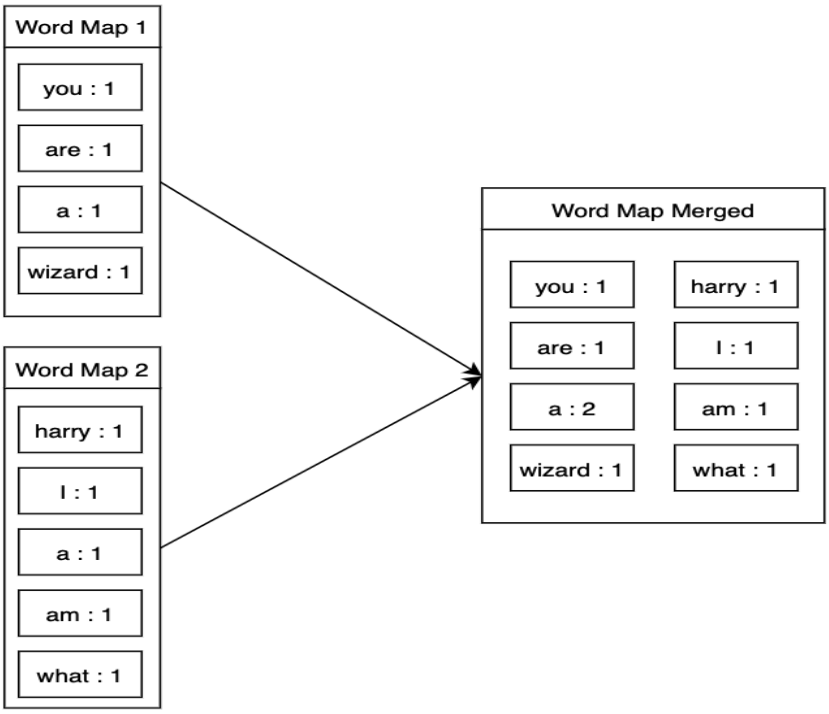
C. The broken words are read from a file and completed using the k-th most frequent word from the trie that matches the word pattern using parallel chunks.

Example:

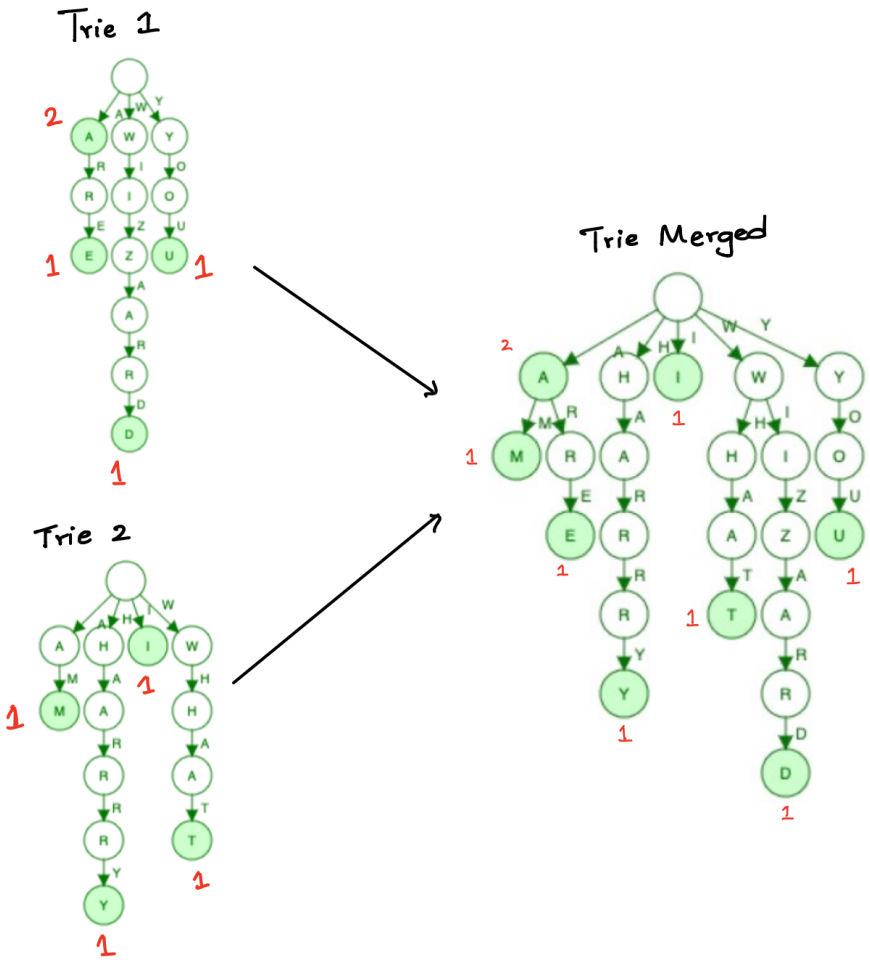
Dictionary built with: *“You are a wizard Harry. I am a what?”*

Broken sentence: *“You ar a wiz har” → “You are a wizard harry”*

Building word frequency dictionary of word chunk size 4



Building and merge tries of chunk size 4
[“you”: 1, “are”: 1, “a”: 2, “wizard”: 1]
[“harry”: 1, “I”: 1, “am”: 1, “what”: 1]



A. Building the word frequency dictionary

1. Words are filtered based on alphabetical presence and converted to lowercase

```
wordCleanerPar :: [[Char]] -> [[Char]]
wordCleanerPar x = L.map (L.map toLower . L.filter isAlpha) x `using` rpar
```

```
wordCleanerPar :: [[Char]] -> [[Char]]
wordCleanerPar x = L.map (L.map toLower . L.filter isAlpha) x `using`
parListChunk 10 rpar
```

```
wordCleanerPar :: [[Char]] -> [[Char]]
wordCleanerPar x = L.map (L.map toLower . L.filter isAlpha) x `using`
parBuffer 10 rdeepseq
```

```
let cw = concat $ runEval (parallelEval wordCleanerPar (chunksOf wSize w))
```

No significant speed up was observed while using parallel logic for word clean-up.

2. Word frequency map built with cleaned up words in parallel

Parallel Eval: Takes a function f and a list as input, and applies the function to every element of the list in parallel using rpar.

```
parallelEval :: (a -> b) -> [a] -> Eval [b]
parallelEval _ [] = return []
parallelEval f (x:xs) =
   do
       y <- rpar (f x)
       ys <- parallelEval f xs
       return (y:ys)
```

Approach 1: Inserting into a map and folding over it

```
mapWordsSeq :: (Foldable t, Ord k) => t k -> Map k Int
mapWordsSeq = L.foldr (\v s->insertWith (+) v 1 s) M.empty
```

```
mergeMapSeq :: (Foldable t, Ord k) => t (Map k Int) -> [(k, Int)]
mergeMapSeq m = M.toList (L.foldr (M.unionWith (+)) M.empty m)
```

Words list is broken down into a chunk of size wSize and (numWords / wSize) maps are built in parallel and the result is available in wordMap which is a list of maps. mSize chunks of maps are taken and merged in parallel. Finally, the remaining small number of maps are merged sequentially.

```
let wordMap   = runEval (parallelEval mapWordsSeq (chunksOf wSize cw))
let wordMerge = runEval (parallelEval mergeMapSeq (chunksOf mSize wordMap))
let wordCount = M.toList (mergeMaps (L.map M.fromList wordMerge))
```

Time taken to print the word frequency map with approach 1

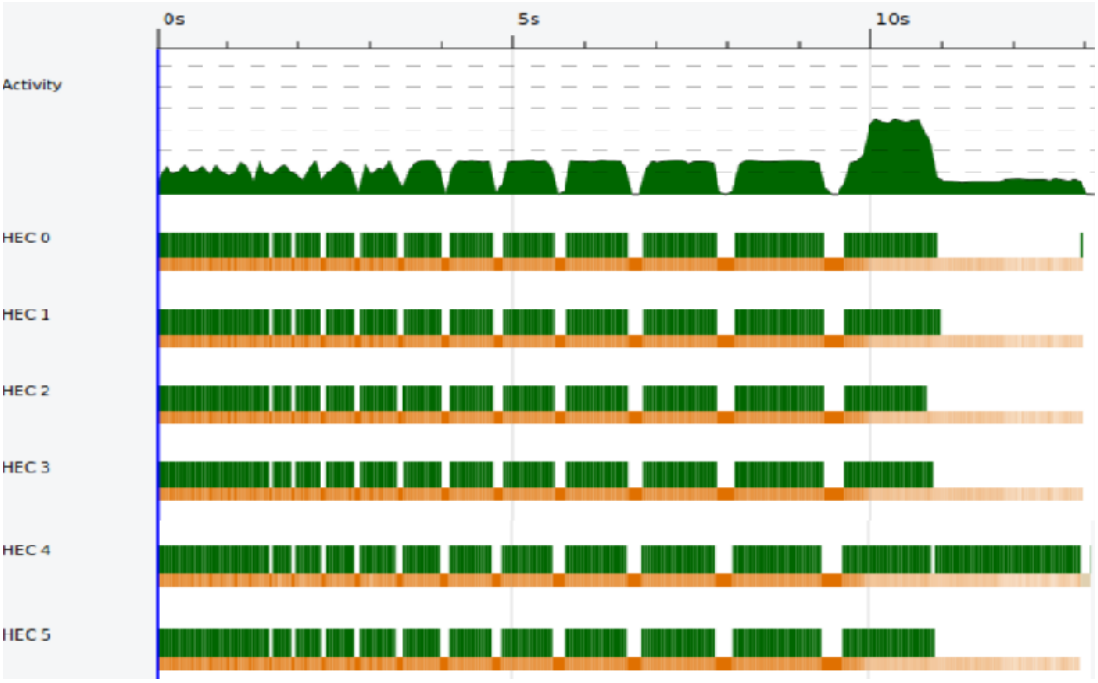
File Size	wSize	mSize	Time (N=2)	Time (N=3)	Time (N=4)	Time (N=6)
5.8 MB	5000	50	1.894 s	1.778 s	1.381 s	1.290 s
84.1 MB	10000	50	21.643 s	19.267 s	17.303 s	13.081 s
161.2 MB	100000	75	59.40 s	50.831 s	39.179 s	33.793 s
322.4 MB	100000	100	122.08 s	108.70 s	83.570 s	59.940 s

File Size	Speedup N=2	Speedup N=3	Speedup N=4	Speedup N=6
5.8 MB	1.13 x	1.20 x	1.56 x	1.66 x
84.1 MB	1.23 x	1.39 x	1.54 x	2.04 x
161.2 MB	1.13 x	1.32 x	1.72 x	1.99 x
322.4 MB	1.12 x	1.26 x	1.64 x	2.28 x

6 core sparks for 84.1 MB file - sequential printing

SPARKS: 1178 (1121 converted, 0 overflowed, 0 dud, 0 GC'd, 57 fizzled)

INIT time 0.001s (0.005s elapsed)
MUT time 18.956s (4.498s elapsed)
GC time 41.126s (9.178s elapsed)
EXIT time 0.000s (0.010s elapsed)
Total time 60.083s (13.692s elapsed)



Threadscope graph utilzing 6 cores for 84.1 MB file - sequential printing

Approach 2: Mapping each word to 1 and mapping to add the counts at the end

```
mapWords :: [String] -> [(String, Int)]
mapWords cw = L.map (\x -> (x,1)) cw
```

```
mapReduce :: (Ord k) => [(k, Int)] -> Map k Int
mapReduce x = (M.fromListWith (+) x)
```

```
mergeMaps :: (Foldable t, Ord k) => t (Map k Int) -> Map k Int
mergeMaps = L.foldr (M.unionWith (+)) M.empty
```

Words list is broken down into a chunk of size wSize and (numWords / wSize) lists with (word, 1) mapping are built in parallel. These maps are reduced in parallel and finally merged to generate the wordCount map.

```
let wordMap = runEval (parallelEval mapWords (chunksOf wSize cw))
let wordMerge = runEval (parallelEval mapReduce wordMap)
let wordCount = M.toList (mergeMaps wordMerge)
```

Time taken to print the word frequency map with approach 2

File Size	wSize	Time N=2	Time N=3	Time N=4	Time N=6
5.8 MB	5000	2.370 s	2.250 s	2.219 s	2.168 s
84.1 MB	10000	27.830 s	25.058 s	21.871 s	21.686 s
161.2 MB	100000	68.430 s	60.090 s	55.870 s	59.542 s
322.4 MB	100000	152.280 s	150.010 s	147.780 s	145.157 s

File Size	Speedup N=2	Speedup N=3	Speedup N=4	Speedup N=6
5.8 MB	0.96 x	1.01 x	1.02 x	1.05 x
84.1 MB	0.97 x	1.08 x	1.24 x	1.25 x
161.2 MB	0.99 x	1.13 x	1.22 x	1.14 x

Approach 1 is significantly faster than approach 2 based on the results.

B. Building the word trie

WordCount which is a list of word and word frequency count pairs which is split into chunks of size mSize. mSize number of tries are built in parallel by using the parallelEval method. These tries are further merged together in parallel of size tSize which in turn is finally merged into one word frequency trie.

```
trieBuilder :: [(String, Int)] -> Trie Int
trieBuilder w = T.fromList (L.map (\x -> (C.pack(fst x), -(snd x))) w)
```

```
mergeTries :: (Foldable t, Num a) => t (Trie a) -> Trie a
mergeTries t = L.foldr (T.mergeBy (\x y -> Just (x + y))) T.empty t
```

```
let wordRes = runEval (parallelEval trieBuilder (chunksOf mSize wordCount))
let trieMerge = runEval (parallelEval mergeTries (chunksOf tSize wordRes))
let trie = mergeTries trieMerge
```

Time taken to print the trie

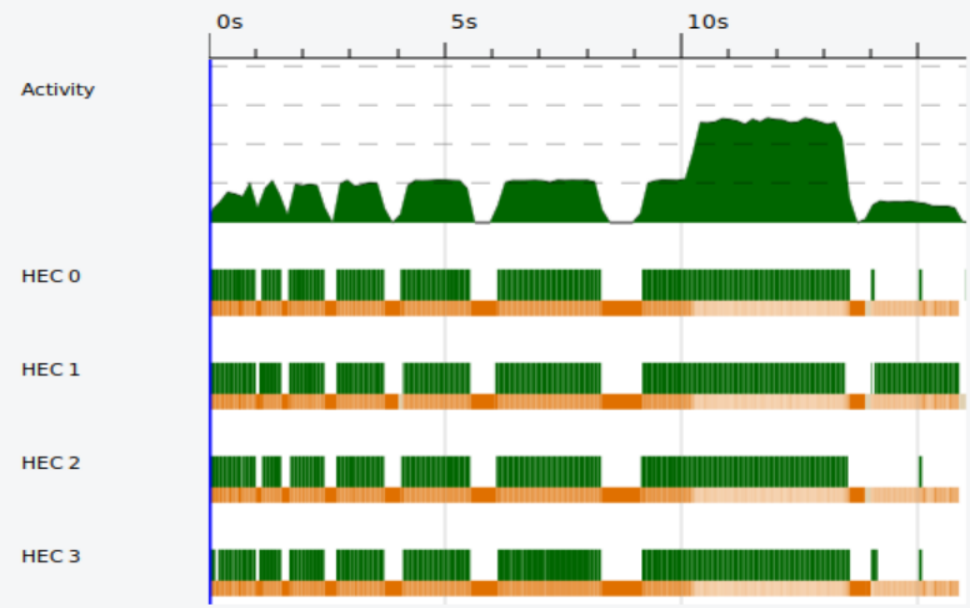
File Size	mSize	tSize	Time N=2	Time N=3	Time N=4	Time N=6
5.8 MB	200	30	2.033 s	1.808 s	1.415 s	1.333 s
84.1 MB	300	50	21.962 s	19.571 s	16.241 s	13.937 s
161.2 MB	300	30	58.972 s	54.656 s	40.528 s	33.254 s
322.4 MB	600	75	115.280 s	109.07 s	91.550 s	69.730 s

File Size	Speedup N=2	Speedup N=3	Speedup N=4	Speedup N=6
5.8 MB	1.20 x	1.35 x	1.73 x	1.82 x
84.1 MB	1.31 x	1.47 x	1.77 x	2.06 x
161.2 MB	1.20 x	1.29 x	1.74 x	2.13 x
322.4 MB	1.22 x	1.29 x	1.54 x	2.02 x

4 core sparks for 84.1 MB file - sequential printing

SPARKS: 209 (191 converted, 0 overflowed, 0 dud, 2 GC'd, 16 fizzled)

INIT time 0.001s (0.008s elapsed)
MUT time 19.660s (4.170s elapsed)
GC time 44.361s (9.634s elapsed)
EXIT time 0.000s (0.008s elapsed)
Total time 64.023s (13.820s elapsed)



Threadscope graph utilzing 4 cores for 84.1 MB file - sequential printing

C. Word Complete

```
getK :: Ord b => [(C.ByteString, b)] -> Int -> [Char] -> [Char]
getK t k w = kWord
   where
       kWord = if length l > k then C.unpack (fst (l !! k !! 0)) else w
       l = groupBy (\x y -> snd x == snd y) L.sortBy (O.comparing snd) t
```

```
wordComplete :: Ord a => [Char] -> Int -> Trie a -> [Char]
wordComplete w k trie = kWord
   where
       resTrie = T.toList (T.submap (C.pack w) trie) `using` rpar
       notPresent = isNothing (T.lookup (C.pack w) trie)
       kWord = if(length resTrie > 0&&notPresent) then getK resTrie k w else w
```

```
completeWords :: Ord a => [[Char]] -> Int -> Trie a -> [[Char]]
completeWords ws k trie = [wordComplete w k trie | w <- ws]
```

Approach 1:

Takes a function, a list to operate on, k, and the built trie and applies the given function to every element of the list in parallel using rpar.

```
parallelComplete :: (t1 -> t2 -> t3 -> a) -> [t1] -> t2 -> t3 -> Eval [a]
parallelComplete _ [] _ _ = return []
parallelComplete f (x:xs) k trie =
   do
       y <- rpar (f x k trie)
       ys <- parallelComplete f xs k trie
       return (y:ys)
```

Split the broken words into chunks of size nc and complete the words in parallel.

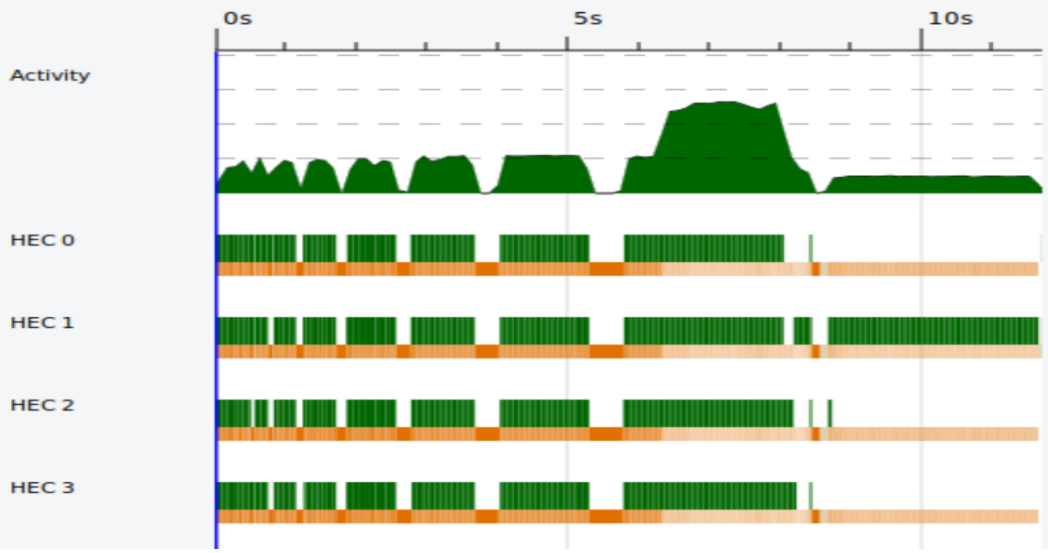
```
let res = concat $ runEval (parallelComplete completeWords (chunksOf nc ocw) k trie)
```

aChunk → clean word chunks
bChunk → frequency map chunks
cChunk → trie building chunks
dChunk → trie merge chunks
eChunk → broken file chunk

Dict File Size	Broken File Size	a, b, c, d	Time (N=2)	Time (N=3)	Time (N=4)	Time (N=6)
5.8 MB	0.004 MB	5000,50,200,30,1000	1.995 s	1.802 s	1.361 s	1.131 s
21.2 MB	0.016 MB	10000,50,300,50,5000	7.054 s	5.412 s	5.031 s	4.755 s
42.2 MB	0.18 MB	20000,50,300,50,5000	16.080 s	14.050 s	12.384 s	12.283 s
74 MB	0.32 MB	100000, 500, 300, 50, 10000	35.721 s	32.197 s	29.740 s	31.395 s

4 core sparks for 42.2 MB file
SPARKS: 452 (391 converted, 0 overflowed, 0 dud, 3 GC'd, 58 fizzled)

INIT time 0.001s (0.008s elapsed)
MUT time 12.214s (4.687s elapsed)
GC time 21.616s (7.003s elapsed)
EXIT time 0.000s (0.004s elapsed)
Total time 33.831s (11.701s elapsed)



Threadscope graph utilzing 4 cores for 42.2 MB file - sequential printing

Approach 2:

Takes a function, a list to operate on, k, built trie, and a cache and applies the given function to every element of the list in parallel using rpar.

```
parallelCompCache :: (t1 -> t2 -> t3 -> t4 -> a) -> [t1] -> t2 -> t3 -> t4 ->
Eval [a]
parallelCompCache _ [] _ _ _ = return []
parallelCompCache f (x:xs) k trie cache =
   do
       y <- rpar (f x k trie cache)
       ys <- parallelCompCache f xs k trie cache
       return (y:ys)
```

Given a list of broken words, k, trie, and the cache, returns the completed words. If a broken word was already seen and completed, then the result is stored in a cache and returned for optimization.

```
compWithCache :: Ord a => [[Char]] -> Int -> Trie a -> Map [Char] [Char] ->
[[Char]]
compWithCache [] _ _ _ = []
compWithCache (w:ws) k trie cache
   | ws == [] = [v]
   | otherwise = v : compWithCache ws k trie nMap
   where
       comp = wordComplete w k trie
       v = if res == "" then comp else res
       nMap = if res == "" then M.insert w comp cache else cache
       res = mapRes w cache
```

Split the broken words into chunks of size nc and complete the words in parallel with cache.

```
let res = concat $ runEval (parallelCompCache compWithCache (chunksOf nc ocw)
k trie M.empty)
```

Approach 2 was significantly slower than approach 1 due to excessive computation and the cache did not speed up the word complete process.

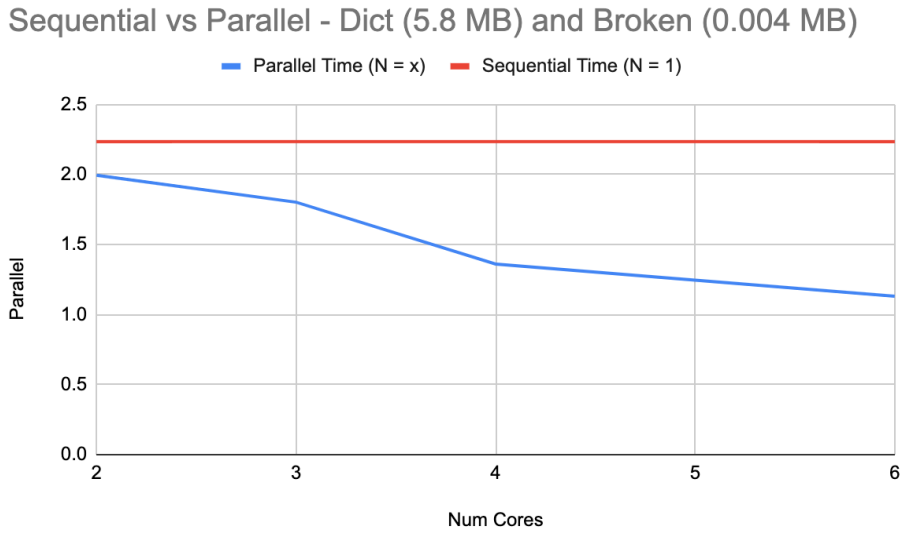
3. Summary

Upto 2x speedup was observed with the parallel implementation and increasing the number of cores led to a significant increase in performance. Better performance can be achieved if perfect chunk sizes are chosen for (a, b, c, d) for the suitable number of cores. Building the wordMap and Trie in parallel yielded significant results as shown in the implementation section. However, printing the output was the sequential bottleneck.

Future considerations: choosing the perfect chunk size for (a, b, c, d) would yield better results. These variables could be further optimized based on the number of cores available. The number of cores can also be increased further to observe greater performance. Garbage collection could also be optimized further based on these chunk sizes. The k-th most frequent word for each broken word could be cached in a parallel hashmap so it doesn't have to be calculated each time. Sorting the words based on the frequency could also be improved by using a highly efficient parallel sorting algorithm.

Dict File Size	Fill File Size	a, b, c, d	Time (N=2)	Time (N=3)	Time (N=4)	Time (N=6)
5.8 MB	0.004 MB	5000,50,200,30,1000	1.995 s	1.802 s	1.361 s	1.131 s
21.2 MB	0.016 MB	10000,50,300, 50,5000	7.054 s	5.412 s	5.031 s	4.755 s
42.2 MB	0.18 MB	20000,50,300, 50,5000	16.080 s	14.050 s	12.384 s	12.283 s
74 MB	0.36 MB	100000, 500, 300, 50, 10000	35.721 s	32.197 s	29.740 s	27.395 s

Dict File Size MB	Fill File Size MB	a, b, c, d	Speed N=2	Speed N=3	Speed N=4	Speed N=6
5.8	0.004	5k,50,200,30,1k	1.12 x	1.24 x	1.64 x	1.98 x
21.2	0.016	10k,50,300,50,5k	1.13 x	1.48 x	1.59 x	1.68 x
42.2	0.18	20k,50,300,50,5k	1.36 x	1.56 x	1.77 x	1.78 x
74	0.36	100k,500,300, 50, 10k	1.20 x	1.33 x	1.43 x	1.56 x



4. Code

Main.hs

```
module Main where

import Lib
import System.Environment(getArgs)
import GHC.Base (VecElem(Int16ElemRep))
import System.IO.Error(catchIOError, isUserError, ioeGetFileName, isDoesNotExistError)
import System.Exit(die)

main :: IO ()
main = do
   [inpFile, kStr, destFile] <- getArgs
   let k = read kStr::Int
   runner inpFile (k-1) destFile

 `catchIOError` \ e -> die $ case ioeGetFileName e of
   _       | isUserError e -> "Usage: autocomplete <dict-filename> <k>
<broken-filename>"
           | otherwise -> show e
```

4.1 Sequential Approach

Lib.hs

```
module Lib
    ( runner, wordCleaner, mapWordsSeq, trieBuilder, completeWords
    ) where

import Data.Char (toLower, isAlpha)
import Data.Map as M
import Data.List as L
import Data.Trie as T
import qualified Data.ByteString.Char8 as C
import Data.Ord as O
import Data.Maybe as Mb
import System.IO(IOMode(WriteMode), hPrint, openFile, hClose)
import System.IO.Error(catchIOError, isUserError, ioeGetFileName, isDoesNotExistError)
import System.Exit(die)

mapWordsSeq :: (Foldable t, Ord k) => t k -> Map k Int
mapWordsSeq = L.foldr (\v s->insertWith (+) v 1 s) M.empty

wordCleaner :: [[Char]] -> [[Char]]
wordCleaner x = L.map (L.map toLower . L.filter isAlpha) x

-- Conventional map reduce
-- mapWords :: [String] -> [(String, Int)]
-- mapWords cw = L.map (\x -> (x,1)) cw
-- mapReduce :: (Ord k) => [(k, Int)] -> Map k Int
-- mapReduce x = (M.fromListWith (+) x)

trieBuilder :: [(String, Int)] -> Trie Int
trieBuilder w = T.fromList (L.map (\x -> (C.pack(fst x), -(snd x))) w)

getK :: Ord b => [(C.ByteString, b)] -> Int -> [Char] -> [Char]
getK t k w = kWord
   where
       kWord = if length l > k then C.unpack (fst (l !! k !! 0)) else w
       l = groupBy (\x y -> snd x == snd y) (L.sortBy (O.comparing snd) t)

wordComplete :: Ord a => [Char] -> Int -> Trie a -> [Char]
wordComplete w k trie = kWord
```

```
   where
       resTrie = T.toList (T.submap (C.pack w) trie)
       notPresent = isNothing (T.lookup (C.pack w) trie)
       kWord = if (length resTrie > 0 && notPresent) then getK resTrie k w else w

completeWords :: Ord a => [[Char]] -> Int -> Trie a -> [[Char]]
completeWords ws k trie = [wordComplete w k trie | w <- ws]

saveFile :: Show a => a -> IO ()
saveFile x = do
   outh <- openFile "output.txt" WriteMode
   hPrint outh x
   hClose outh

runner :: FilePath -> Int -> FilePath -> IO ()
runner inpFile k destFile = do
   content <- readFile inpFile
   let w = words content
   let cw = wordCleaner w

   let wordCount = M.toList (mapWordsSeq cw)
   let trie = trieBuilder wordCount

   dest <- readFile destFile
   let ow = words dest
   let ocw = wordCleaner ow

   let res = completeWords ocw k trie
   saveFile res

 `catchIOError` \ e -> die $ case ioeGetFileName e of
   Just fn | isDoesNotExistError e -> "autocomplete: " ++ fn ++ ": openFile: does not
exist (No such file or directory)"
   _       | isUserError e -> "Usage: autocomplete <dict-filename> <k>
<broken-filename>"
           | otherwise -> show e
```

4.2 Parallel Approach

Lib.hs

```
module Lib
    ( runner, wordCleaner, mapWordsSeq, trieBuilder, completeWords
    ) where

import Data.Char (toLower, isAlpha)
import Data.Map as M
import Data.List as L
import Data.Trie as T
import Control.Parallel.Strategies
import qualified Data.ByteString.Char8 as C
import Data.Ord as O
import Data.List.Split (chunksOf)
import Data.Maybe as Mb
import System.IO(IOMode(WriteMode), hPrint, openFile, hClose)
import System.IO.Error(catchIOError, isUserError, ioeGetFileName, isDoesNotExistError)
import System.Exit(die)

parallelEval :: (a -> b) -> [a] -> Eval [b]
parallelEval _ [] = return []
parallelEval f (x:xs) =
    do
        y <- rpar (f x)
        ys <- parallelEval f xs
        return (y:ys)

parallelComplete :: (t1 -> t2 -> t3 -> a) -> [t1] -> t2 -> t3 -> Eval [a]
parallelComplete _ [] _ _ = return []
parallelComplete f (x:xs) k trie =
    do
        y <- rpar (f x k trie)
        ys <- parallelComplete f xs k trie
        return (y:ys)

-- Parallel complete with cache -- Approach 2
{-
parallelCompCache :: (t1 -> t2 -> t3 -> t4 -> a) -> [t1] -> t2 -> t3 -> t4 -> Eval [a]
parallelCompCache _ [] _ _ _ = return []
parallelCompCache f (x:xs) k trie cache =
```

```
   do
       y <- rpar (f x k trie cache)
       ys <- parallelCompCache f xs k trie cache
       return (y:ys)
-}

-- Different order reduce
mapWordsSeq :: (Foldable t, Ord k) => t k -> Map k Int
mapWordsSeq = L.foldr (\v s->insertWith (+) v 1 s) M.empty

mergeMapSeq :: (Foldable t, Ord k) => t (Map k Int) -> [(k, Int)]
mergeMapSeq m = M.toList (L.foldr (M.unionWith (+)) M.empty m)

-- Conventional Map reduce -- Approach 2
{-
mapWords :: [String] -> [(String, Int)]
mapWords cw = L.map (\x -> (x,1)) cw

mapReduce :: (Ord k) => [(k, Int)] -> Map k Int
mapReduce x = (M.fromListWith (+) x)
-}

mergeMaps :: (Foldable t, Ord k) => t (Map k Int) -> Map k Int
mergeMaps = L.foldr (M.unionWith (+)) M.empty

wordCleaner :: [[Char]] -> [[Char]]
wordCleaner x = L.map (L.map toLower . L.filter isAlpha) x

trieBuilder :: [(String, Int)] -> Trie Int
trieBuilder w = T.fromList (L.map (\x -> (C.pack(fst x), -(snd x))) w)

mergeTries :: (Foldable t, Num a) => t (Trie a) -> Trie a
mergeTries t = L.foldr (T.mergeBy (\x y -> Just (x + y))) T.empty t

getK :: Ord b => [(C.ByteString, b)] -> Int -> [Char] -> [Char]
getK t k w = kWord
   where
       kWord = if length l > k then C.unpack (fst (l !! k !! 0)) else w
       l = groupBy (\x y -> snd x == snd y) (L.sortBy (O.comparing snd) t)

wordComplete :: Ord a => [Char] -> Int -> Trie a -> [Char]
wordComplete w k trie = kWord
```

```
    where
        resTrie = T.toList (T.submap (C.pack w) trie)
        notPresent = isNothing (T.lookup (C.pack w) trie)
        kWord = if (length resTrie > 0 && notPresent) then getK resTrie k w else w

completeWords :: Ord a => [[Char]] -> Int -> Trie a -> [[Char]]
completeWords ws k trie = [wordComplete w k trie | w <- ws]

-- Approach 2 with cache
{-
mapRes :: Ord k => k -> Map k [Char] -> [Char]
mapRes w cache = Mb.fromMaybe "" (M.lookup w cache)

compWithCache :: Ord a => [[Char]] -> Int -> Trie a -> Map [Char] [Char] -> [[Char]]
compWithCache [] _ _ _ = []
compWithCache (w:ws) k trie cache
    | ws == [] = [v]
    | otherwise = v : compWithCache ws k trie nMap
    where
        comp = wordComplete w k trie
        v = if res == "" then comp else res
        nMap = if res == "" then M.insert w comp cache else cache
        res = mapRes w cache
-}

saveFile :: Show a => a -> IO ()
saveFile x = do
    outh <- openFile "output.txt" WriteMode
    hPrint outh x
    hClose outh

runner :: FilePath -> Int -> FilePath -> IO ()
runner inpFile k destFile = do
    content <- readFile inpFile
    let w = words content
    let cw = wordCleaner w

    let aChunk = 20000
    let bChunk = 50
    let cChunk = 300
    let dChunk = 50
    let eChunk = 5000
```

```
  let wordMap   = runEval (parallelEval mapWordsSeq (chunksOf aChunk cw))
  let wordMerge = runEval (parallelEval mergeMapSeq (chunksOf bChunk wordMap))
  let wordCount = M.toList (mergeMaps (L.map M.fromList wordMerge))

  -- Conventional map reduce -- Approach 2
  -- let wordMap = runEval (parallelEval mapWords (chunksOf 10000 cw))
  -- let wordMerge = runEval (parallelEval mapReduce wordMap)
  -- let wordCount = M.toList (mergeMaps wordMerge)

  let wordRes = runEval (parallelEval trieBuilder (chunksOf cChunk wordCount))
  let trieMerge = runEval (parallelEval mergeTries (chunksOf dChunk wordRes))
  let trie = mergeTries trieMerge

  dest <- readFile destFile
  let ow = words dest
  let ocw = wordCleaner ow

  -- With a word cache -- Approach 2
  -- let res = concat $ runEval (parallelCompCache compWithCache (chunksOf 5000 ocw)
k trie M.empty)

  let res = concat $ runEval (parallelComplete completeWords (chunksOf eChunk ocw) k
trie)
  saveFile res

 `catchIOError` \ e -> die $ case ioeGetFileName e of
  Just fn | isDoesNotExistError e -> "autocomplete: " ++ fn ++ ": openFile: does not
exist (No such file or directory)"
  _       | isUserError e -> "Usage: autocomplete <dict-filename> <k>
<broken-filename>"
          | otherwise -> show e
```

4.3 Testing

Spec.hs

```
import Data.Map as M
import qualified Lib as L
import Data.Trie as T

t1 :: [[Char]] -> [[Char]] -> [Char]
t1 w res = if res == L.wordCleaner w then "t1 passed" else "t1 failed"

t2 :: (Foldable t, Ord k) => t k -> [(k, Int)] -> [Char]
t2 w res = if res == (M.toList (L.mapWordsSeq w)) then "t2 passed" else "t2 failed"

t3 :: Ord a => [[Char]] -> Trie a -> Int -> [[Char]] -> p -> p -> p
t3 ws trie k res tex1 tex2 = if res == (L.completeWords ws k trie ) then tex1 else
tex2

main :: IO ()
main = do
   print(t1 ["*ab", "HelLo", "world"] ["ab", "hello", "world"])
   print(t2 ["hi", "hi"] [("hi", 2)])

   let trie = L.trieBuilder [("youre", 1), ("a", 1), ("wizard", 2), ("wizards", 1),
("harry", 3), ("hardware", 1)]
   print(t3 ["youre", "a", "wiz", "har"] trie 0 ["youre", "a", "wizard", "harry"] "t3
passed" "t3 failed")
   print(t3 ["youre", "a", "wiz", "har"] trie 1 ["youre", "a", "wizards", "hardware"]
"t4 passed" "t4 failed")
```

4.4 Build Instructions

stack build

./autocomplete-exe [WORD_DICT_FILE] [k] [BROKEN_FILE] +RTS -ls -s -N4

Example:
./autocomplete-exe dict.txt 1 broken.txt +RTS -ls -s -N4
./autocomplete-exe dict.txt 1 broken.txt +RTS -ls -s -N1

stack test

- Includes basic tests to check frequency counts and trie autocomplete
- Tested against Shakespeare file set for frequency counts

5. References

https://stackoverflow.com/questions/2895748/how-to-write-list-to-file
https://en.wikipedia.org/wiki/MapReduce
https://hackage.haskell.org/package/bytestring-trie-0.2.6/docs/Data-Trie.html
https://hackage.haskell.org/package/split-0.2.3.4/docs/Data-List-Split.html#v:chunksOf